Final Exam — ECE 438

Due May 16^{th}, 2020

Name:

David Kirby

- This is a open-book, open-note exam.
- You *must* work on this exam by yourself!
- Calculators are permitted, but no communication devices of any kind are allowed.
- Each question is marked with its number of points.
- Show your answers in the space provided for them. Write neatly and be well organized.
- Show your work if you want to get credit!

Good luck!

Problem	Maximum	Score
1	15	15
2	15	15
3	20	19
4	20	19
5	20	19
6	15	14
7	15	15
Total	120	116

1. **(15 points)** You are looking to accelerate public-key encryption and found through simulation that 95% of the execution time of a particular algorithm is spent performing modular multiplication. Assume for the following problems that the execution time of your algorithm is 2 seconds without hardware acceleration.

   (a) What would be the minimum achievable execution time of this same algorithm if you developed hardware that could accelerate just the modular multiplication by 50x? Calculate the overall speedup.

   $t_{new} = \frac{2s \times 0.95}{50x} + 2s \times (1 - 0.95) = 0.138s$

   $\text{Speedup} = \frac{1}{\frac{0.95}{50x} + (1 - 0.95)} = 14.493x$

   (b) What would be the execution time and speedup achievable with a 20x acceleration of modular multiply?

   $t_{new} = \frac{2s \times 0.95}{20x} + 2s \times (1 - 0.95) = 0.195s$

   $\text{Speedup} = \frac{1}{\frac{0.95}{20x} + (1 - 0.95)} = 10.256x$

   (c) What is the name given to the design principle used in parts a) and b)?

   Amdahl's Law

   (d) What would the theoretical maximum speedup be for your program if you could infinitely accelerate modular multiply?

   $t_{new} = \frac{2s \times 0.95}{\infty} \to 0 + 2s \times (1 - 0.95) = 0.1s$

   $\text{Speedup} = \frac{2s}{0.1s} = 20x$

2. **(15 points)** Imagine you have a 1GHz RISC processor with split L1 instruction and data caches, each 16kB in size with a hit time of 1ns. Access to main memory takes 50 ns, and 35% of instructions access memory. The L1 instruction cache miss rate is 1%, while the L1 data cache miss rate is 4%.

   (a) Calculate the Average Memory Access Time (AMAT) for each of the L1 caches.

   $\text{AMAT}_{\text{I\$}} = 1\text{ns} + 0.01 \times 50\text{ns} = 1.5\text{ns}$
   $\text{AMAT}_{\text{D\$}} = 1\text{ns} + 0.04 \times 50\text{ns} = 3\text{ns}$

   (b) Assuming your processor has a CPI of 1.3 with an ideal memory hierarchy, what is the CPI considering memory stalls?

   $\text{CPI} = 1.3\frac{\text{cc}}{\text{instr}} + 0.01 \times 50\frac{\text{cc}}{\text{instr}} + 0.04 \times 0.35 \times 50\frac{\text{cc}}{\text{instr}} = 2.5\frac{\text{cc}}{\text{instr}}$

   (c) You are considering the inclusion of a 256kB L2 cache to improve your performance. If the miss rate of the L2 is 35%, what would the AMAT of the L1 instruction and data caches be with the L2 cache? Assume a 6ns L2 hit time.

   $\text{AMAT}_{\text{L2}} = 6\text{ns} + 0.35 \times 50\text{ns} = 23.5\text{ns} = 23.5\text{cc}$
   AMAT_{L2} is the new miss penalty of the L1 caches.
   $\text{AMAT}_{\text{I\$}} = 1\text{ns} + 0.01 \times 23.5\text{ns} = 1.235\text{ns}$
   $\text{AMAT}_{\text{D\$}} = 1\text{ns} + 0.04 \times 23.5\text{ns} = 1.94\text{ns}$

   (d) What would the CPI of your processor be with the L2 cache? What is the speedup due to the L2 cache?

   $\text{CPI} = 1.3\frac{\text{cc}}{\text{instr}} + 0.01 \times 23.5\frac{\text{cc}}{\text{instr}} + 0.04 \times 0.35 \times 23.5\frac{\text{cc}}{\text{instr}} = 1.864\frac{\text{cc}}{\text{instr}}$
   $\text{Speedup} = \frac{2.5}{1.864} = 1.3412\text{x}$

3. **(20 points)** For the following problem, assume a 5-stage pipelined processor with a branch delay slot and branch resolution in the Execute stage. Consider the code below:

```
    addi $t2, $t1, 48
loop:
    lw $t4, 40 ($t1)
    lw $t5, 80 ($t1)
    add $t6, $t4, $t5
    sw $t6, 120 ($t1)
    bne $t1, $t2, loop
    addi $t1, $t1, 4
    add $t2, $t1, $t3
```

(a) Draw the pipeline execution diagram for the first *two* iterations of the above code when an "assume not taken" branching scheme is used.

Clock cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
`addi $t2, $t1, 48`	F	D	E	M	W															
`lw $t4, 40 ($t1)`		F	D	E	M	W														
`lw $t5, 80 ($t1)`			F	D	E	M	W													
`add $t6, $t4, $t5`				F	D	D	E	M	W											
`sw $t6, 120 ($t1)`					F	F	D	E	M	W										
`bne $t1, $t2, loop`							F	D	E	M	W									
`addi $t1, $t1, 4`								F	D	E	M	W								
`lw $t4, 40 ($t1)`									X	F	D	E	M	W						
`lw $t5, 80 ($t1)`											F	D	E	M	W					
`add $t6, $t4, $t5`												F	D	D	E	M	W			
`sw $t6, 120 ($t1)`													F	F	D	E	M	W		
`bne $t1, $t2, loop`															F	D	E	M	W	
`addi $t1, $t1, 4`																F	D	E	M	W

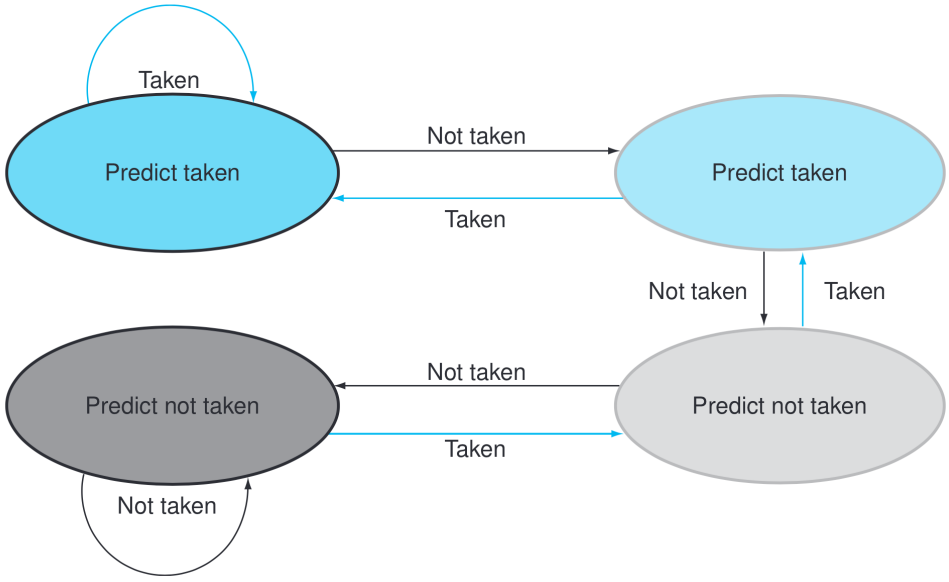
(b) How many clock cycles are required to execute the above code to completion when an "assume not taken" branching scheme is used?

We have one cycle for the first `addi`. Then we have eleven loop iterations in which the `bne` is "Taken," each requiring eight cycles (see diagram above). Then we have the last loop iteration in which the `bne` is "Not Taken," requiring seven cycles (no flushing). We then have four cycles to drain the pipeline (i.e. to finish the `bne`) and finally one cycle to complete the last `add` instruction. Therefore, the number of clock cycles required is:
$1 + 11 \times 8 + 7 + 4 + 1 = 101$ clock cycles.

perfect!

(c) How many clock cycles are required to execute the above code assuming a 100% correct branch predictor.

A 100% correct branch predictor means no flushes, therefore, the number of clock cycles required is:
$1 + 12 \times 7 + 4 + 1 = 90$ clock cycles.



(d) Consider the use of the two-bit predictor shown above. Assuming the predictor starts in the bottom right state (*different than homework*), how many clock cycles are required to execute the above code?

If the predictor is in the bottom right state (weakly Not Taken) when entering the loop, it will predict not taken for the first execution of `bne` which will move the predictor state to the top right (weakly Taken). On the next iteration, it will then move to the top left state (strongly Taken) and remain there until the execution of the last `bne`, in which case it will move to the top right (weakly Taken) state. Therefore, the predictor will predict "Not Taken" for the first iteration, then "Taken" for every other execution of the `bne`. With two extra cycles introduced due to misprediction during the first and last iterations of the loop, the number of required cycles is:
$1+12\times 7+2+4=92$ clock cycles.

(e) What is the accuracy of this predictor given the code above? Compare that to the accuracy of the "assume not taken" scheme.

In the case of our loop, we predict correctly 10 out of 12 times or $\frac{10}{12}=83\%$. Contrast that with the "assume not taken" scheme which correctly predicts 1 out of 12 times or $\frac{1}{12}=8\%$.

(f) What speedup does the branch predictor from part (d) provide over the "assume not taken" scheme?

Speedup $=\frac{101}{92}=1.098$x

4. **(20 points)** The following questions are related to pipelining and instruction level parallelism:

   (a) Compare the i486 pipeline with that of the MIPS pipeline we studied in lecture. How does it differ? How is it similar? How did the performance compare?

   The most obvious difference between the i486 and the MIPS pipelines is in the ISA - i486 uses a CISC instruction set while MIPS uses RISC. Our MIPS supported branch delay allowing for jumps to execute in one cycle, whereas the x86 series had no concept of branch delay, it wasn't built into the ISA. The i486 was, however, able to execute instructions in one clock cycle, similar to the MIPS pipeline we studied. This was an advance over the i386 and was accomplished by integrating the cache into the pipeline.

   -1 so it wasn't perfectly 1 instr/cc use spec to compare

   (b) How did the Intel Pentium achieve a performance benefit over the i486? How did the performance compare?

   The Pentium is 8-micron technology and essentially smashed two i486 pipelines together. This superscalar pipelining fetched more instructions per cycle. The i486 also used a unified cache while the Pentium used split caches for the instructions and the data.

   The SPEC ratio on the i486 was about 20 (at 33MHz) while the Pentium achieved 60 at 66MHz, comparable to the R4000 and three times speedup over the i486.

   perfect!

   (c) Describe a tournament predictor. How does it provide a benefit over a local or global predictor?

   The tournament predictor uses a meta predictor to decide between a local predictor and a global predictor and it uses that meta predictor to keep track of which predictor is predicting better.

   (d) Define precise exceptions. How do we maintain precise exceptions in an in-order pipeline? How do we do so in an out-of-order machine?

   Precise exceptions associate an exception with an instruction (i.e. determines exactly which instruction was the offending instruction) and involves completing all instructions before the exception and flushing any instructions that occur after the exception as though they never began execution.

   After precisely associating an exception with an instruction, in-order pipelines will draw a line in the sand and allow instructions before that offending instruction to commit state (write to the memory) while all instructions, including the offending instruction and all instructions after, are actually flushed and prevented from writing any state.

   For out-of-order machines, the instruction fetch and decode unit uses Reservation Stations to buffer everything into Functional Units which then execute the instructions out of order to speed things along. These executed instructions are then put back into order by a reorder buffer before they are allowed to commit to memory.

   very good!

5. **(20 points)** The following questions are related to cache design and memory hierarchy:

   (a) Briefly describe the types of cache misses talked about in lecture. Remember the three C's.

   Compulsory: These are the initial misses due to an empty (cold) cache.
   Conflict: These are misses due to a rigid block placement strategy (i.e. low associativity).
   Capacity: These are misses due to the cache being too small to hold the entire working set of data and instructions.

   (b) How does adding multiple levels to the memory hierarchy improve performance?

   Adding multiple levels to the memory hierarchy increases the chances of a cache hit. Increasing the cache size reduces capacity misses.

   (c) Why might one add *associativity* to a cache? How might one add associativity? What are the drawbacks to set-associative caches?

   Adding associativity reduces conflict misses by providing a more flexible block placement strategy. For example, direct mapped has only one location in the cache that a block can be placed, but full associativity allows blocks to be placed anywhere in the cache.

   The drawbacks to associativity include increased complexity, access (hit) time, and power as well as added difficult when pipelining the cache.

   (d) Why might the instruction cache in a *split cache* system have a lower miss rate than the data cache?

   Instruction cache in a split cache system exhibits a lot more spatial and temporal locality compared to data access cache.

   (e) Describe the purpose of a *write buffer*. Where in the memory hierarchy might you place a write buffer?

   The write-buffer allows you to prioritize reads over writes. A write-buffer can improve the performance by not stalling the processor while data is being written to the next slower level. In the case of write-through, writes go to the buffer from the processor and avoid processor stalls as long as the write buffer is not full. In the case of write-back, writes go into the buffer when a block is being evicted allowing the cache controller to immediately begin retrieving the requested cache block. (see Figure 1, page 8)

   (f) Briefly describe *write-back* and *write-through* cache write policies. When might one be preferred over the other?

   Write-back only writes data to the next level of the memory hierarchy when a dirty block is being evicted, whereas write-through updates the next level of the memory hierarchy on every write. Write-back adds a little more complexity to our cache, but reduces traffic to the next level of the hierarchy. Write-through is simple to implement and keeps the next level of memory consistent, but is a major performance bottleneck because the processor must stall on multiple writes and

creates congestion on the bus and next level of memory.

Write-back is almost always preferred; however, write-through might be preferred in the case of a shared level of cache as the constant updates keep the memory consistent.

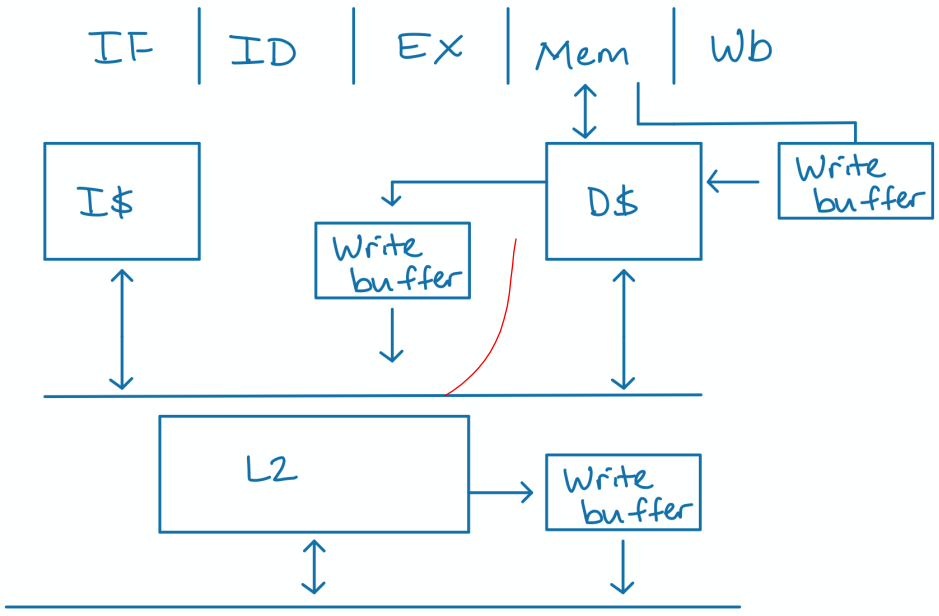


Figure 1: Write Buffer for Question 5e

(g) Describe each of the following architectures: Von Neumann, Harvard, Modified Harvard.

Von Neumann architecture stores instructions and data in the same memory, while Harvard architecture stores instructions and data in separate memories. A modified Harvard architecture is where we go from a split cache at one level to a unified cache at the next level. This contrasts with a complete Harvard architecture where, at the programmer level, you split your instructions and data into separate memories.

6. **(15 points)** The following questions are related to virtual memory:

   (a) What was the primary reason virtual memory was developed?

   Virtual memory was developed to cope with small amounts of main memory, swapping data between disk and main memory automatically. exactly!

   (b) What are other advantages virtual memory provides?

   Other advantages of virtual memory include:

   - Allowing multiple processes to share resources easily
   - Each process or program has its own virtual view of memory
   - Protection between various processes and between processes and the OS

   (c) What is the purpose of the TLB? How do the TLB and L1 cache work together?

   Operating systems use some form of Least Recently Used (LRU) to determine what goes into physical memory versus disk. The translation lookaside buffer (TLB) acts as a small cache (typically fully associative) that stores the most recently used page table entries in order to speed things up. It is managed by the operating system. The TLB works with the L1 cache by mapping the virtual page number that translates to a physical address. In my mind, this scenario is like a NAT in networking, where we map local IP addresses to public-facing IP addresses.

   (d) What role does the Operating System (OS) play when it comes to virtual memory? What additional hardware support is required to run an OS?

   The OS manages page tables and the TLB, which keeps track of address mapping. Storing pages on disk allows a program's virtual address space to be larger than the machines physical address space. Additional hardware support required to run an OS include User and Kernel modes and address regions. & exceptions → 1

7. **(15 points)** Assuming a direct-mapped cache with a byte-address that is broken up such that bits 0-2 are for the byte offset, bits 3-5 are for the word offset, bits 6-14 are for the index, and bits 15-31 are for the tag, answer the following questions:

   (a) How large are the words in this machine?

   $2^{\text{byte offset}} = 2^3 = 8$ bytes $= 64$ bits.

   (b) How many words are in each cache block? How many bytes are in each cache block?

   $2^{\text{word offset}} = 2^3 = 8$ words in each cache block.
   8 words $\times$ 8 bytes $= 64$ bytes in each cache block.

   (c) How many cache blocks are in the cache? How many sets are in the cache?

   $2^{\text{index}} = 2^9 = 512$ sets in the cache, and because it is direct-mapped, there are also 512 blocks.

   (d) How large is the data store of this cache?

   $512 \times 64 = 32$KiB

   (e) How large is the tag store? Assume a *valid* and *dirty* bit are included with each tag.

   Each tag is 19 bits and there are 512 tags $\rightarrow$ 19 bits $\times$ 512 tags $= 9728$ bits.

   (f) If you were to modify the cache to be 4-way set associative (*different from homework*) but keep the data store the same size, what size would the tag and index be? How large would the tag store be?

   The index would decrease by two bits while the tag would increase by two bits. Thus the index would be 8 bits and the tag would be 20 bits (including valid and dirty bits). The number of tags is the same because each block needs a tag. The tag store would be 20 bits $\times$ 512 tags $= 10240$ bits.

   you subtracted added/bit 6yx said 2 so 04

   (g) How much memory can the above machine address?

   The address is 32 bits in length so this machine can address 4GiB.